Janita Kaski

Full Stack -verkkosovelluksen suunnittelu ja toteutus

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

30.4.2024

Tiivistelmä

Tekijä:	Janita Kaski
Otsikko:	Full Stack -verkkosovelluksen suunnittelu ja toteutus
Sivumäärä:	38 sivua
Aika:	30.4.2024
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ohjaajat:	Osaamisaluejohtaja, Janne Salonen

Insinöörityön tarkoituksena oli suunnitella ja toteuttaa verkkosovellus hevosten ylläpitoon liittyvien tapahtumien hallintaan sekä aikataulun suunnitteluun. Työn aihe valikoitui tarpeesta saada käyttöön sovellus, johon voisi keskitetysti tallentaa tärkeimmät tiedot ja asiakirjat, pitää esimerkiksi valmennus- tai kilpailupäiväkirjaa, sekä lisätä hoitotoimenpiteisiin liittyviä merkintöjä. Tavoitteena oli kehittää verkkosovellus, jossa voisi hallita tätä kokonaisuutta yhdessä paikassa ja jonka toiminnallisuudet sekä käyttökokemus vastaavat työn tarkoitusta.

Verkkosovellus kehitettiin käyttäen Django-verkkokehystä, joka perustuu Python-ohjelmointikieleen tarjoten kyseisen sovelluksen suunnitteluun ja toteutukseen sopivat työkalut ja tekniikat. Sovelluksen taustajärjestelmä sekä palvelinpuolen ja käyttöliittymän ominaisuudet ja vuorovaikutus suunniteltiin Djangon projektirakenteen mukaisesti. Toteutuksen edetessä sovelluksen toiminnallisuudet kehittyivät ja muotoutuivat yhteen Djangon arkkitehtuurimallia noudattaen. Työn toteutuksen edetessä testattiin sovelluksen toiminnallisuuksia loppukäyttäjän näkökulmasta, ja sitä että käyttökokemus vastaa asetettua tavoitetta.

Tuloksena saavutettiin toimiva verkkosovellus, jolla on selkeä ja looginen rakenne, sekä sujuva vuorovaikutus taustajärjestelmän, palvelinpuolen ja käyttöliittymän välillä, mikä oli kehittämistehtävän ensisijainen tavoite. Sovelluksen käytettävyys osoittautui miellyttäväksi ja suorituskyky vastasi asetettuja tavoitteita, toimintojen reagoidessa nopeasti käyttäjän pyyntöihin. Työn tulosten perusteella voidaan päätellä, että Django tarjoaa tehokkaat työkalut monipuolisten verkkosovellusten kehittämiseen ja ylläpitoon, sekä mahdollisuuden sisällyttää monipuolisesti myös valmiita käyttöliittymäkomponentteja käyttökokemuksen parantamiseksi.

Tulosten hyödynnettävyys ulottuu laajemmalle kuin vain hevosiin liittyvän tiedon hallintaan. Sovelluksen toiminnallisuudet ovat helposti sovellettavissa erilaisiin käyttökohteisiin. Lopputuote tarjoaa vahvan pohjan sovelluksen jatkokehitykselle ja laajentamiselle, ja työn tuloksia voidaan hyödyntää myös muissa projekteissa, jotka vaativat vastaavanlaista tiedonhallintaa ja -tallentamista verkkosovelluksissa.

Avainsanat: Full Stack, verkkosovellus, verkkosovelluskehitys, Django, Python, JavaScript

Abstract

Author:	Janita Kaski
Title:	Design and Implementation of a Full Stack Web Application
Number of Pages:	38 pages
Date:	30 April 2024
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communication Technology
Supervisors:	Janne Salonen, Competence Area Director

The purpose of the engineering thesis was to design and implement a web application for managing and scheduling activities related to horse care and training. The topic was chosen based on the need for a centralized application to store important data and documents, maintain training or competition diaries, and store care-related notes. The goal was to develop a web application where users can effectively manage this entity on a single platform with clear functions and a pleasant user experience.

The web application was developed using the Django web framework, which is based on the Python programming language and provides suitable tools and techniques for designing and implementing such applications. The application's backend, server-side, and user interface features and interactions were designed following Django's project structure. As the implementation progressed, the application's functionalities evolved and integrated in accordance with Django's architecture. Throughout the implementation, the application's functionalities were tested from an end-user perspective to ensure that the user experience aligned with the defined goals.

The outcome was a functional web application with a clear and logical structure and interaction between the backend, server-side, and user interface, which was the primary goal of the development. The usability of the application proved to be smooth, and its performance met the established goals by responding quickly to user requests, achieved using JavaScript in the user interface implementation. Based on the project’s result, it can be concluded that Django provides efficient tools for developing and maintaining versatile web applications.

The usability of the results extends beyond managing information related to horses. The application's functionalities can be easily adapted to various use cases. The final product provides a strong foundation for further development and expansion of the application, and its findings can be leveraged in other projects that require equivalent data storage and management in web applications.

Keywords: Full Stack, Web application, Web development, Django, Python, JavaScript

Sisällys

1 Johdanto

Insinöörityön tarkoitus on suunnitella ja toteuttaa verkkosovellus hevosten ylläpitoon liittyvien tapahtumien hallintaan sekä aikataulun suunnitteluun. Työn aihe valikoitui tarpeesta saada käyttöön sovellus, johon voisi keskitetysti tallentaa tärkeimmät tiedot ja asiakirjat, pitää esimerkiksi valmennus- tai kilpailupäiväkirjaa, sekä lisätä hoitotoimenpiteisiin liittyviä merkintöjä. Monesti tiedot ovat useassa paikassa hajallaan tai muistin varassa, ja niitä voi olla hankala etsiä jälkeenpäin. Tavoitteena on kehittää verkkosovellus, jossa voisi hallita tätä kokonaisuutta keskitetysti, ja jonka toiminnallisuudet olisivat selkeitä ja käytön kannalta olennaisia. Työn kohderyhmänä ovat hevosenomistajat tai -ylläpitäjät.

Työn toteutustavaksi valikoitui Django-verkkokehys, sillä Django tarjoaa erinomaiset työkalut ja ominaisuudet verkkosovellusten kehittämiseen, pohjautuen Python-ohjelmointikieleen. Tutkimuskysymykset liittyvät Djangon tekniikoiden ympärille, keskittyen verkkosovelluksen eri osa-alueiden vuorovaikutukseen ja siihen millainen sovelluksen käytettävyyden ja suorituskyvyn tulisi olla. Ensimmäinen kysymys on, miten projektiin valittujen tekniikoiden avulla on mahdollista toteuttaa mahdollisimman selkeä verkkosovelluksen taustajärjestelmän rakenne sekä palvelinpuolen suoraviivainen toimintalogiikka. Toinen kysymys on, miten käyttöliittymän ominaisuudet tukevat sovelluksen toiminnallisuuksia siten, että käyttökokemus olisi mahdollisimman miellyttävä ja onnistunut.

Tavoiteltavana tuloksena on verkkosovellus, jonka ominaisuudet ja rakenne vastaavat työn tarkoitusta ja tavoitetta, mikä olisi myös hyvä perusta jatkaa tulevaisuudessa sovelluksen kehittämistä laajemmaksi tuotteeksi. Sovelluksen julkaiseminen ei sisälly tämän työn aihealueeseen, vaan työ on rajattu suunnittelu- ja toteutusvaiheisiin. Työssä käytettävä tutkimusmenetelmä keskittyy vahvasti omaan kehittämistyöhön, verkkosovelluksen suunnittelun ja toteutuksen edetessä, korostaen käytännönläheistä ja soveltavaa lähestymistapaa. Pyrin soveltamaan teoreettista tietoa käytäntöön ja kehittämään sopivia ratkaisuja ja tekniikoita saavuttaakseni työn tavoitteen mukaisen lopputuloksen.

2 Tietoperusta

2.1 Full Stack -verkkosovelluskehitys

Termi *Full Stack* tarkoittaa verkkosovelluksen suunnittelua, rakenteellista määrittelyä ja toteuttamista alusta loppuun. Tämä pitää sisällään verkkosovelluksen käyttöliittymän sekä taustajärjestelmän kehittämisen. Sana ”stack” viittaa erilaisiin tekniikoihin, joilla kaikilla on oma roolinsa ja merkityksensä verkkosovellusympäristössä. Käyttöliittymä on se sovelluksen osa, jonka kanssa käyttäjä on suoraan vuorovaikutuksessa verkkoselaimen kautta. Sovelluksen taustajärjestelmä kattaa sovelluksen tietokannan ja toimintalogiikan palvelinpuolella. Käyttäjät eivät suoraan näe tätä kokonaisuutta, mutta käyttöliittymä puolestaan kommunikoi taustajärjestelmän kanssa jatkuvasti, lähettäen ja vastaanottaen tietoa palvelimelta. Täten kaikkien verkkosovelluksen osa-alueiden tulee toimia yhdessä ja kehittyä suhteessa toisiinsa.

Verkkosovelluksen käyttöliittymä, verkkosivustona ajateltuna, kattaa sovelluksen ulkoasun eli visuaalisen puolen, kuten asettelun ja värimaailman. Käyttöliittymäsuunnittelun päätavoitteena on kuitenkin tehdä vuorovaikutuksesta käyttäjän ja sovelluksen välillä tarkoituksenmukaista ja miellyttävää. Tähän liittyy läheisesti käyttökokemus, joka yhdistää käyttöliittymän sovelluksen taustajärjestelmään, johon sisältyy sivuston toiminnallisuudet palvelinpuolella ja se, millaiseksi käyttäjä kokee sovelluksen ominaisuudet ja käytön.

Käyttöliittymän tulisi esittää informaatio ja toiminnot siten, että käyttäjä voi nopeasti hahmottaa, mitä verkkosivuilla tulisi tehdä ja miten toimia. Verkkosivujen sisällön tulee olla järjestetty loogisesti ja helposti navigoitavalla tavalla. Esteettinen käyttöliittymä antaa käyttäjälle positiivisen vaikutelman ja lisää käytön miellyttävyyttä. Käyttäjien tulee haluta käyttää sovellusta uudelleen ja sen tulee olla merkityksellinen ja helppokäyttöinen. Hyvän tasapainon löytäminen on kuitenkin tärkeää, kuten Ghidersa (2023) toteaa puhuttaessa estetiikasta, tuote voi olla täydellinen, mutta jos kukaan ei avaa sitä useammin kuin kerran, se ei ole käyttökelpoinen.

Käyttöliittymän visuaalisten elementtien toteutuksen keskiössä on kolme eri tekniikkaa, *Hypertext Markup Language* (HTML), *Cascading Style Sheets* (CSS) sekä *JavaScript* (Myers 2020). HTML koostuu elementeistä, jotka kuvaavat verkkosivun sisältöä, kuten otsikot, kappaleet ja linkit. HTML luo verkkosivun rakenteen, kun taas CSS kertoo kuinka rakenne tulisi esittää. CSS määrittää, miten elementit näytetään verkkosivuilla, kuten sivun asettelu, värit ja fontit.

Verkkoselain muodostaa verkkosivusta HTML-dokumenttiobjektimallin (*Document Object Model* eli DOM). DOM määrittelee HTML-elementit, niiden ominaisuudet ja käyttötavat. DOM-metodit mahdollistavat JavaScriptin ja HTML-dokumentin välisen vuorovaikutuksen, mikä tekee mahdolliseksi sivun dynaamisen päivittämisen. (JavaScript HTML DOM.) JavaScript voi käsitellä ja muuttaa kaikkia HTML-dokumentin elementtejä. JavaScript-ohjelmointikieli on keskeisessä osassa suunniteltaessa tehokkaita ja asynkronisia toimintoja käyttöliittymässä, eli tapahtumia, jotka toteutuvat ilman verkkosivujen uudelleenlatausta. JavaScript voi esimerkiksi muokata sivun sisältöä ja reagoida käyttäjän toimintoihin välittömästi ilman tietoa palvelimelta, tai vastavuoroisesti se voi hakea tietoja palvelimelta ja päivittää sivun sisältöä niiden perusteella.

Taustajärjestelmässä käytettäviä palvelinpuolen ohjelmointikieliä ovat esimerkiksi Python, Java tai PHP. Palvelinpuolen tehtävänä on hallita tietokantojen käyttöä tietojen tallentamiseen ja hakemiseen sekä määrittää, mitä tietoja lähetetään käyttäjälle. Taustajärjestelmä vastaa siis käyttäjien verkkosivuilla lähettämiin pyyntöihin vuorovaikutuksessa tietokantojen kanssa, ohjaten tiedonkulkua kokonaisvaltaisesti palvelimen ja käyttöliittymän välillä (Introduction to the server side).

Jokaisella ohjelmointikielellä on omat vahvuutensa ja valinta perustuu esimerkiksi verkkosovellusprojektin vaatimuksiin. Sovelluskehityksessä käytetään usein verkkokehyksiä, jotka tarjoavat valmiin rakenteen verkkosovelluksen kehittämiseen, sekä standardit ja työkalut, jotka auttavat sovelluskehittäjiä helpottamaan ja nopeuttamaan ohjelmointityötä. Kehittäjät voivat valita sopivan verkkokehyksen sen mukaan, millä ohjelmointikielellä he haluavat työskennellä tai millaisia tekniikoita he tarvitsevat sovelluksensa kehittämiseen.

2.2 Django-verkkokehys

Django-verkkokehyksen rakenne ja toiminnallisuus perustuvat Python-kieleen. Python on korkean tason yleiskäyttöinen ohjelmointikieli, jolla on yksinkertainen ja johdonmukainen syntaksi ja suuri standardikirjasto (General Python FAQ). Djangon filosofia korostaa selkeän ja luettavan koodin merkitystä, ja kuten Dinder (2022) kertoo, luettavuus onkin olennainen syy, miksi Python valittiin Djangon perustaksi. Python on myös helppo integroida muihin kieliin, mikä tekee siitä yhden suositummista kielistä verkkokehittäjien keskuudessa (Top 7 Programming Languages for Backend Web Development). Suosiosta kertoo myös TIOBE Index (2024), joka on ohjelmointikielten suosion indikaattori. Huhtikuun 2024 listauksen mukaan Python on suosituin ohjelmointikieli, vuoden seuranta-ajalla. Suosion laskenta perustuu ammattitaitoisten kehittäjien ja kolmansien osapuolten tarjoamien palveluiden määrään maailmanlaajuisesti.

Django-projekti toteutetaan kehitysympäristössä, joka tarjoaa työkalut verkkosovelluksen kehittämiseen ja testaamiseen, eli jossa voi kirjoittaa ja muokata koodia. On olemassa useita ympäristöjä, jotka ovat erinomaisia Django-projektien kehittämiseen, esimerkiksi Visual Studio Code (VS Code), joka on kevyt ja monipuolinen tekstieditori. VS Code tarjoaa laajan valikoiman laajennuksia Python-kehitykseen sekä monia käteviä ominaisuuksia, kuten koodin vihjeitä ja virheiden korostusta (Setting up Visual Studio Code).

Kun Django-projekti luodaan kehitysympäristössä, muodostuu kuvan 1 mukainen hakemisto, joka koostuu projektin oletustiedostoista. Juurihakemiston (kuvassa ’mysite’) alla on komentorivityökalu ”manage.py”, jonka avulla projektin kanssa voi olla vuorovaikutuksessa. Juurihakemisto sisältää sisäisen hakemiston, joka on varsinainen Python-paketti projektille. Tiedosto ”__init__.py” on tyhjä tiedosto, joka määrittää, että hakemisto on Python-moduuli. Muut tiedostot sisältävät Django-projektin asetuksia ja resursseja. (Writing your first Django app, part 1.)

```
mysite/
    manage.py
    mysite/
        __init__.py
        settings.py
        urls.py
        asgi.py
        wsgi.py
```

Kuva 1. Django-projektin hakemisto (Writing your first Django app, part 1).

Django-projekti tarvitsee vähintään yhden Django-sovelluksen (app). Django-sovellus on eri asia kuin verkkosovellus kokonaisuutena tai käsite ”sovellus”, josta yleisesti puhutaan. Django-sovellus on Python-paketti, joka noudattaa tiettyjä käytäntöjä, ja niitä hallitaan projektin sisällä. Sovellusten määrään vaikuttaa se, miten laaja projekti on kyseessä ja onko verkkosovelluksessa useita toimintoja, jotka eivät suoraan liity toisiinsa. Tällöin hallinta voi olla helpompaa, jos itsenäisiä sovelluksia on useita saman projektin sisällä.

Kun Django-sovellus luodaan, muodostuu projektin hakemistoon sovelluksen oletustiedostot. Nämä tiedostot tarjoavat pohjan sovelluksen palvelinpuolen koodaukselle ja kehitykselle. Kuva 2 näyttää Djangon automaattisesti luomat tiedostot, jotka löytyvät sovelluksen nimellä nimetyn hakemiston alta. Huomaa, että kuvan tiedosto ”urls.py” on kuitenkin luotu hakemistoon manuaalisesti, ja tämä on eri kuin Django-projektin urls.py-tiedosto. Projektin tiedosto määrittelee URL-reitityksen projektin tasolla, eli tiedosto ohjaa pyynnöt Djangon eri sovelluksiin. Sovelluksen urls.py-tiedosto taas määrittää URL-reitit kyseisen sovelluksen sisällä.

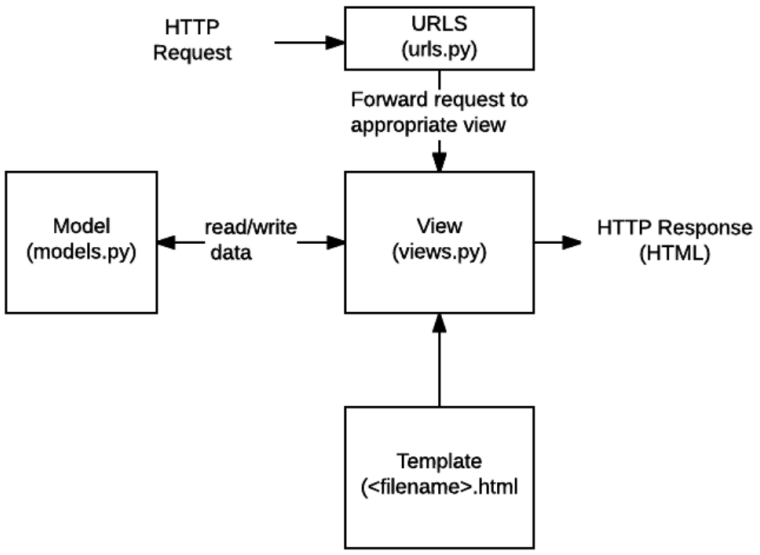
```
__init__.py
admin.py
apps.py
migrations/
    __init__.py
models.py
tests.py
urls.py
views.py
```

Kuva 2. Django-sovelluksen hakemisto (Writing your first Django app, part 1).

Tiedostossa ”views.py” määritellään niin kutsutut näkymät, jotka vastaavat käyttäjän verkkosivustolla lähettämien pyyntöjen käsittelystä ja vastausten palauttamisesta. Tiedostossa ”models.py” luodaan sovelluksen tietorakenteet ja tietokantataulut. Nämä kaksi tiedostoa ovat pääosassa Djangon arkkitehtuurimallissa, jonka avulla luodaan selkeä ja jäsennelty koodipohja, joka helpottaa verkkosovelluksen kehitystä, testausta ja ylläpitoa.

2.3 Djangon arkkitehtuurimalli

Djangon arkkitehtuuri perustuu *Model-View-Template* (MVT) -malliin, joka toimii sovelluksen rakenteena ja mahdollistaa erilaisten ratkaisujen toteuttamisen projektin tarpeiden mukaisesti. Kuva 3 havainnollistaa, miten kommunikointi käyttäjän ja sovelluksen välillä etenee Djangon MVT-mallissa. Käyttäjän lähettäessä *HyperText Transfer Protocol* (HTTP) -pyynnön verkkoselaimessa, pyyntö ohjautuu URL-reitityksen (URLS) kautta tietyn näkymän käsiteltäväksi. HTTP on protokolla, jonka avulla verkkoselaimet kommunikoivat verkkopalvelimien kanssa. Näkymä (View) kommunikoi mallin (Model) kanssa noutaakseen tietoja tai tallentaakseen, päivittääkseen tai poistaakseen niitä tietokannasta. Kun näkymä on saanut tarvittavat tiedot, se palauttaa sisällön pohjan (Template) kautta verkkoselaimelle HTTP-vastauksena. (Django introduction.)



Kuva 3. Käyttäjän lähettämän HTTP-pyynnön eteneminen verkkopalvelimella ja Djangon MVT-arkkitehtuurimallissa (Django introduction).

Mallit ja tietokanta

Mallit edustavat tietokantaa ja ne kuvaavat tietokantataulujen rakennetta ja sisältöä. Malleihin määritellään tietokentät ja niiden ominaisuudet. (Django Models.) Djangon *Object-Relational Mapping* (ORM) kääntää Python-koodin tietokantakyselyiksi automaattisesti ja toimii siltana mallien ja tietokannan välillä. Kun tietokantakerros, eli luokat, kentät ja niiden suhteet on määritelty, tietokanta suorittaa *Structured Query Language* (SQL) -kyselyn ja palauttaa tuloksen, jonka ORM kääntää takaisin Python-arvoiksi. Tämä ei ainoastaan nopeuta kehitysprosessia, vaan myös vähentää virheiden mahdollisuutta ja tekee koodista selkeämpää. (Buelta 2022; Shaw ym. 2023.)

SQL on ohjelmointikieli, joka tarjoaa standardoidun tavan kommunikoida relaatiotietokantojen kanssa, kuten SQLite. Django käyttää SQLite-tietokantaa oletuksena, ja kun projekti luodaan, tietokanta muodostuu automaattisesti. SQLite on kevyt relaatiotietokanta ja se on osa Pythonin vakiokirjastoja (Dinder 2022).

Mallit ovat Python-luokkia, ja luokkien välillä voi olla monenlaisia yhteyksiä. Djangon ORM mahdollistaa erilaisten kenttien käytön malleissa tietokantasuhteiden määrittelyyn. ManyToOne-suhde määritellään käyttämällä ForeignKey-kenttää, joka luo yhteyden tietueeseen toisessa mallissa. ManyToManyField-suhde tarkoittaa, että useampi mallin tietue voi viitata moneen tietueeseen toisessa mallissa ja päinvastoin. OneToOneField-kenttä taas luo yksisuuntaisen yhteyden kahden mallin välille, jolloin tietue yhdessä mallissa voi viitata tarkalleen yhteen tietueeseen toisessa mallissa. (Dinder 2022.)

Kun sovelluksen mallit suhteineen ja ominaisuuksineen on luotu, Djangolla on kaikki tiedot, joita se tarvitsee muodostaakseen sovelluksen tietokannan. Python-koodin muuntamista tietokantarakenteiksi kutsutaan migraatioksi, jolloin Django luo vastaavat taulukot SQLite-tietokantaan mallimäärittelyjen perusteella. Lisäksi jos mallien sisältöä muutetaan, tulee migraatio suorittaa. Migraatiotiedosto on Python-skripti, joka sisältää komentoja siitä, miten tietokantataulut tulisi muokata vastaamaan uusia tai muuttuneita malleja. Migraatiotiedostot tallentuvat migrations-kansioon sovelluksen hakemistossa. (Writing your first Django app, part 2.)

Näkymät ja käyttöliittymätekniikat

Näkymät ovat vuorovaikutuksessa sovelluksen tietokannan, eli mallien ja käyttöliittymän kanssa, kuten Djangon arkkitehtuurimallista nähdään. Näkymät käsittelevät käyttäjän verkkoselaimella lähettämät pyynnöt URL-reitityksen avulla ja lopuksi palauttavat vastauksen, joka voi olla esimerkiksi HTML-sivu, JSON-vastaus tai virheilmoitus, jossa hyödynnetään pohjia (templates). Pohjat ovat HTML-tekstitiedostoja, ja ne hahmontavat näkymien sisällön ja määrittelevät verkkosivujen rakenteen ja asettelun. Pohjat mahdollistavat niin kutsutun esitystason erottamisen sovelluksen toimintalogiikasta muuttamatta taustalla olevaa Python-koodia. Tämä heijastaa Djangon tapaa virtaviivaistaa verkkokehitystä.

Pohjiin voidaan sisällyttää CSS- ja JavaScript-tekniikoita, jotka lisäävät verkkosivujen interaktiivisuutta ja estetiikkaa. Pohjiin voi linkittää JavaScript-tiedostoja

tai ulkoisia käyttöliittymäkirjastoja, mikä auttaa pitämään koodin siistinä ja eriytettynä HTML:stä. (Django introduction.) Esimerkiksi CSS-pohjainen kehys Bootstrap on tarkoitettu verkkosivujen visuaalisuuden suunnitteluun hyödyntäen valmiita tyylittelyratkaisuja HTML-, CSS- ja JavaScript-komponenttien avulla. Bootstrap sisältää useita jQuery-laajennuksia ja edellyttää siksi yleensä jQueryn käyttöä. (Myers 2020.) jQuery on JavaScript-kirjasto, joka helpottaa kommunikaatiota DOM:n ja JavaScriptin välillä (Frontend vs Backend).

JavaScript-koodia voi myös kirjoittaa suoraan pohjiin käyttämällä luokkia tai funktioita. JavaScript-funktiot ovat koodilohkoja, jotka suorittavat tietyn tehtävän, kun funktiota kutsutaan. Funktiot auttavat jäsentämään koodia, mikä helpottaa suurten koodikantojen ymmärtämistä ja ylläpitoa. (Chidera 2023.) Käyttämällä kumpaa tahansa menetelmää, luokkia tai funktioita, tehtäviä voi toistaa monta kertaa ilman, että tarvitsee kirjoittaa koodia uudelleen. Kuten Simpson (2023) toteaa, tärkeintä on johdonmukaisuus projektissa.

Näkymien ja pohjien välinen tiedonvaihto helpottuu *JavaScript Object Notation* (JSON) -vastausten avulla. Näkymät tarkastelevat prosessitietoja, esimerkiksi lisäävät sisältöä käyttäjän tallentaminen lomakkeiden kautta, ja palauttavat JSON-objekteja, joita JavaScript-toiminnot käyttävät sitten päivittääkseen käyttöliittymän tai antaakseen palautetta käyttäjälle. JSON on tekstimuoto tietojen tallentamiseen ja siirtämiseen. JSON-vastauksen palauttaminen tarkoittaa, että näkymä lähettää tiedot takaisin asiakkaalle JSON-merkkijonon muodossa. (JSON - Introduction.)

JavaScript tarjoaa kaksi tapaa tehdä asynkronisia HTTP-pyyntöjä palvelimelle. *Ajax* (Asynchronous JavaScript ja XML) -tekniikka mahdollistaa verkkosivuston päivittämisen ilman sivun uudelleenlatausta sekä reaaliaikaisen tiedonsiirron palvelimen ja selaimen välillä. Ajaxin avulla JavaScript voi lähettää ja vastaanottaa tietoja palvelimelta taustalla ja päivittää sivun sisältöä dynaamisesti käyttäjän toimintojen perusteella. Fetch-funktio perustuu XMLHttpRequest-objektiin kuten Ajax, mutta suurin ero on, että se käyttää niin kutsuttuja lupauksia (promises), mikä tarjoaa paremman luettavuuden ja ylläpidettävyyden. Se voi käsitellä JSON-dataa käyttäen response.json()-metodia, kun taas Ajax tarvitsee

JSON.parse()-funktion muuntaakseen vastauksen JSON-merkkijonon JavaScript-objektiksi. (JavaScript Fetch Vs Ajax.)

Django on myös suunniteltu suojaamaan verkkosivustoa automaattisesti, joka estää yleisiä tietoturvaongelmia (Django Web Framework). Esimerkiksi Cross Site Request Forgery (CSRF) -suojaus varmistaa, että jokainen pyyntö, joka saapuu sovellukseen, on aito ja sen on lähettänyt oikea käyttäjä. Jos sovellus esimerkiksi käsittelee lomakkeiden lähetystä ja sitä kautta tietokantatietojen päivittämistä, CSRF-suojaus näkymissä sekä HTML- ja JavaScript-koodissa varmistavat, että vain valtuutetut pyynnöt voivat suorittaa näitä toimintoja. (Cross Site Request Forgery protection.) Kuten Ghidersa (2023) toteaa, turvallisuus määritellään pitkälti sen mukaan, mikä on sovelluksen käyttötarkoitus ja laajuus.

Kuten huomataan, arkkitehtuuri on laaja käsite, jonka määrittäminen helpottaa verkkosovelluksen kehittäjää saamaan kokonaiskuvan siitä, miten sovelluksen eri osat ja komponentit tulee yhdistää ja miten ne kommunikoivat keskenään. Arkkitehtuurissa sekä toiminnallisen että luovan suunnittelun tulisi kulkea rinnakkain. Kehitysprosessin tulisi keskittyä alusta alkaen käyttökokemukseen loppukäyttäjän näkökulmasta, jotta toivottu lopputulos saavutetaan. Kuten Myers (2020) toteaa, on tärkeää miettiä, mitä käyttäjä odottaa näkevänsä näytöllä ennen jokaista vuorovaikutusta, sen aikana ja sen jälkeen.

Varhaisessa vaiheessa tehdyllä arkkitehtuurilla on merkittäviä etuja. Muutokset ovat helpompia toteuttaa suunnitteluvaiheessa kuin myöhemmin toteutusvaiheessa. Kun peruselementit ovat vakaat, teknisen toteutuksen aikana voidaan keskittyä yksityiskohtiin ja toteutukseen ilman suuria muutoksia sovelluksen rakenteessa, mikä edistää sujuvaa ja tehokasta kehitysprosessia. Tämä lähestymistapa auttaa varmistamaan projektin onnistumisen ja laadukkuuden.

3 Tekninen toteutus

3.1 Taustajärjestelmä

Djangon projektirakenne sekä arkkitehtuurimalli toimivat verkkosovelluksen suunnittelun ja toteutuksen pohjana. Verkkosovellus on toteutettu yhtenä Django-sovelluksena. Koin tämän riittäväksi, sillä sovelluksen toiminnallisuudet liittyvät tiivisti yhteen. Koodaustyön aloitin malleista, tiedostossa models.py, joihin määrittelin sovelluksen tietorakenteet. Jokaiseen sovelluksen osa-alueeseen liittyy oma mallinsa, joille määrittelin kenttiin liittyvät ominaisuudet ja rajoitukset. Kentät kuvaavat tietoja, joita malli tallentaa tietokantaan, ja oli tärkeää määrittää, että vain olennaista ja tarpeellista tietoa tallennetaan. Malleja olisi helppo muokata ja laajentaa jälkeenpäin, jos toimintoja lisätään.

Sovelluksen käyttö alkaa hevosen lisäämisellä tietokantaan, eli hevosen tietojen tallentamisella. Täten ensimmäinen malli, eli Python-luokka, on ”Horse” (hevonen). Tämä malli kuvaa hevosen perustietojen tallennusta, jonka perusteella muodostuu hevoselle profiili. Esimerkkikoodi 1 näyttää tuonnin (import) joka on pakollinen sisällyttää models.py-tiedoston alkuun, koska se antaa pääsyn Djangon ORM:ään ja sen luokkiin, kuten ”Model” ja ”CharField”. Ilman tätä tuontia Python ei tunnistaisi näitä luokkia eikä voisi tulkita niitä, eikä malleja voisi määritellä tai käyttää ORM:n tarjoamia toimintoja, kuten tietokantakyselyitä.

Jokainen Horse-mallin kenttä vastaa tietokantataulukon saraketta, joka tallentaa mallin esiintymät Field-luokan mukaan, ja sulkuihin sisällytetään tarvittavat muut määrittelyt. Kentät ovat valinnaisia täyttää, lukuun ottamatta ensimmäistä kenttää ”name”, eli hevosen nimi on pakollinen syöttää tallennettaessa hevosen tietoja. Kentissä on rajoituksia, kuten merkkijonojen enimmäispituus. Metodi ”__str__” on Python-luokkien erityinen menetelmä, joka määrittää, että luokan esiintymät esitetään merkkijonoina. Metodi palauttaa siis Horse-objektin merkkijonoesityksen, joka sisältää hevosen nimen.

```
from django.db import models


class Horse(models.Model):
    name = models.CharField(max_length=100)
    breed = models.CharField(max_length=100, null=True, blank=True)
    gender = models.CharField(max_length=100, null=True, blank=True)
    colour = models.CharField(max_length=100, null=True, blank=True)
    birth_date = models.DateField()
    description = models.TextField(default='Default description')
    health_details = models.TextField(default='Default description')


    def __str__(self):
        return self.name
```

Esimerkkikoodi 1. Horse-malli hevosen tietojen tallentamiseksi tietokantaan.

Sovelluksen lomakeluokat määrittelin tiedostoon ”forms.py”. Django-lomake perii forms.ModelForm-luokan, joka on suunniteltu toimimaan yhdessä Django-mallien kanssa. Esimerkkikoodi 2 näyttää lomakeluokan, joka määrittelee mitkä tietokentät näkyvät käyttäjälle verkkosivulle avautuvassa lomakkeessa. Meta-luokka määrittelee, että tämä lomake käyttää Horse-mallia ja ”fields” kertoo lomakkeeseen sisällytettävät kentät. Kaikkia malliin määritettyjä kenttiä ei tarvitse sisällyttää lomakkeeseen. Esimerkkikoodi näyttää myös tuonnit, jotka tämä tiedosto tarvitsee, jotta luokat tunnistetaan ja tulkitaan oikein.

```
from django import forms
from .models import Horse

class HorseForm(forms.ModelForm):
    class Meta:
        model = Horse
        fields = ['name', 'gender', 'birth_date', 'description']
```

Esimerkkikoodi 2. Horse-mallia käyttävä lomakeluokka tiedostossa forms.py.

Kun käyttäjä tallentaa hevoselle profiilin, sovellus luo hevoselle automaattisesti päiväkirjan. Esimerkkikoodin 3 Diary (päiväkirja) -malli on yhdistetty Horse-malliin OneToOneField:n avulla, mikä tarkoittaa, että jokainen päiväkirja liittyy täsmälleen yhteen hevoseen ja että jokaisella hevosella on enintään yksi päiväkirja. Päiväkirjan tarkoitus ei ole välttämättä pitkien tekstien kirjoittaminen vaan pikemminkin tärkeimpien merkintöjen tallentaminen.

Koodiin on sisällytetty myös signaalivastaanottaja, joka liittyy sekä Horse- että Diary-malliin. Kun uusi hevonen tallennetaan tietokantaan, tämä toiminto luo uuden päiväkirjaesiintymän, joka liittyy hevosen profiiliin. Tämä varmistaa, että mallit ”Horse” ja ”Diary” pysyvät synkronoituina tietokannassa.

```
from django.db.models.signals import post_save
from django.dispatch import receiver


class Diary(models.Model):
    horse = models.OneToOneField(Horse, on_delete=models.CASCADE,
            related_name='diary')

    def __str__(self):
        return self.horse.name


@receiver(post_save, sender=Horse)
def create_or_update_horse_diary(sender, instance, created, **kwargs):
    if created:
        Diary.objects.create(horse=instance)
    else:
        instance.diary.save()
```

Esimerkkikoodi 3. Diary-malli on yhdistetty Horse-malliin. Signaali luo hevoselle päiväkirjan automaattisesti aina kun uusi hevonen tallennetaan.

Käyttäjä voi luoda hevosen profiilissa päiväkirjamerkinnän, joka tallentuu hevosen päiväkirjaan. Esimerkkikoodiin 4 sisällytetty DiaryEntry (päiväkirjamerkintä) -mallin viiteavain ”ForeignKey” luo yhteyden Diary-malliin, eli jokainen päiväkirjamerkintä liittyy tiettyyn päiväkirjaan. Määrittely ”models.CASCADE” tarkoittaa, että hevosen profiilin poiston yhteydessä poistuu tietokannasta myös hevosen päiväkirja.

Malli havainnollistaa eri kategoriavaihtoehdot, jotka käyttäjä voi valita lomakkeella. Kategorian ideana on lajitella päiväkirjamerkinnät, jotta tiedon hallinta olisi helpompaa. Vaihtoehtoja voi laajentaa myöhemmin useampaan eri aiheeseen. Malli mahdollistaa yhden liitetiedoston lisäyksen päiväkirjamerkinnän yhteyteen. Liite voi olla esimerkiksi eläinlääkärinlausunto tai kilpailupöytäkirja. Esimerkkikoodiin sisällytetty DiaryEntryFile (päiväkirjamerkinnän liite) -malli liittyy DiaryEntry-malliin, ja se mahdollistaa useamman liitteen tallentamisen jo

aiemmin tallennetun päiväkirjamerkinnän oheen. Tämä antaa joustavuutta myös liitteiden käsittelyyn erillään päiväkirjamerkinnästä, kuten niiden poistamiseen.

```
class DiaryEntry(models.Model):

    CATEGORY_CHOICES = [
        ('training', 'Valmennus'),
        ('competition', 'Kilpailut'),
        ('health', 'Terveys'),
        ('other', 'Muu'),
    ]

    diary = models.ForeignKey(Diary, on_delete=models.CASCADE,
            related_name='entries')
    date = models.DateField()
    title = models.CharField(max_length=200)
    content = models.TextField()
    category = models.CharField(max_length=50,
               choices=CATEGORY_CHOICES)
    file = models.FileField(upload_to='diary_entries/',
           null=True, blank=True)

    is_archived = models.BooleanField(default=False)

    def __str__(self):
        return f"{self.date} - {self.title}"


class DiaryEntryFile(models.Model):
    diary_entry = models.ForeignKey(DiaryEntry,
                  related_name='files', on_delete=models.CASCADE)
    file = models.FileField(upload_to='diary_entries/')

    def __str__(self):
        return self.file.name
```

Esimerkkikoodi 4. DiaryEntry-malli päiväkirjamerkinnän tallentamiseen. DiaryEntryFile-malli mahdollistaa liitteiden lisäämisen tallennettuun merkintään.

Events (tapahtumat) -malli mahdollistaa kalenteritapahtumien tallentamisen tietokantaan. Kun uusi tapahtuma luodaan tietokantaan, Django luo automaattisesti uniikin id:n ja käyttää sitä tapahtuman yksilöimiseen tietokannassa. Tämä malli on liitetty Horse-malliin, eli aina kun uusi hevonen tallennetaan tietokantaan, hevosen nimi tulee valittavaksi kalenteritapahtuman kirjauksessa, jotta tapahtuma voidaan kohdistaa tarvittaessa tietylle hevoselle. Tämä mahdollistaa tapahtumien tallentamisen kalenteriin hevoskohtaisesti, jotta tulevia tapahtumia voidaan suunnitella ilman päällekkäisyyksiä aikatauluissa.

```
class Events(models.Model):
    id = models.AutoField(primary_key=True)
    name = models.CharField(max_length=255,null=True,blank=True)
    start = models.DateTimeField(null=True,blank=True)
    end = models.DateTimeField(null=True,blank=True)
    horse = models.ForeignKey('Horse',
            on_delete=models.CASCADE, null=True, blank=True)
    description = models.TextField(null=True, blank=True)
```

Esimerkkikoodi 5. Events-malli mahdollistaa hevoskohtaisten kalenteritapahtumien tallentamisen.

Kun sovelluksen mallit oli määritelty, tai kun niitä piti tarkentaa, suoritin migraatiot, jonka avulla Django teki muutokset tietokantaan. Täten malleja pystyi helposti tarkentamaan projektin edetessä, ilman että tietokantataulukoita tarvitsisi poistaa tai luoda uusia. Näin ollen tietokanta oli koko kehityksen ajan helposti päivitettävissä ja reaaliaikainen. Seuraava sovelluskehityksen vaihe oli suunnitella sovelluksen toimintojen logiikka sekä käyttöliittymä mallien pohjalta.

3.2 Käyttöliittymä

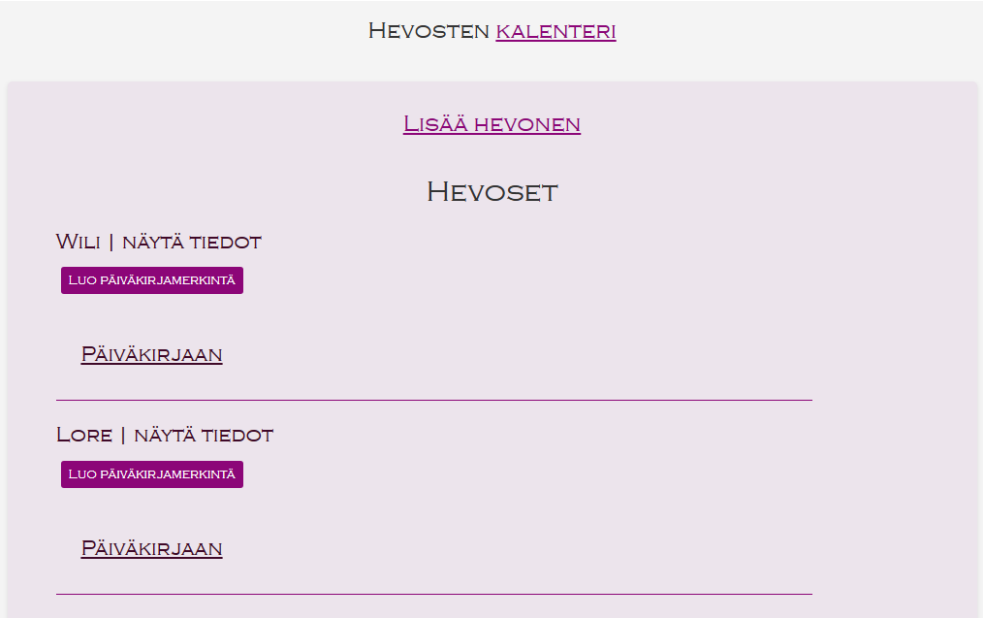
Teknisen toteutuksen käyttöliittymän osuus kattaa pohjat, eli HTML-tekstitiedostot ja niihin sisällytettävän JavaScript-koodin sekä sovelluksen näkymät palvelinpuolella URL-reitityksineen. Käyttöliittymällä viittaan enemmän siihen kokonaisuuteen, miten käyttäjä on vuorovaikutuksessa sovelluksen kanssa, kuin käyttöliittymän ulkoisiin ominaisuuksiin. Näkymät (tiedostossa ”views.py”) kommunikoivat verkkosivuston kanssa, vastaten käyttäjän pyyntöihin, olemalla yhteydessä sovelluksen taustajärjestelmän malleihin. Sovelluksen toimintalogiikkaa suunnitellessa, pidin ensisijaisen tärkeänä, että toiminnot olisivat suoraviivaisia ja sovellusta olisi sujuvaa käyttää, ilman tarvetta ohjeistaa käyttäjää eri toiminnallisuuksien löytämiseksi. Käyttöliittymän estetiikkaa on helppo muokata ja parantaa, mutta jos sovelluksen rakenne ja toiminnallisuudet ovat epäloogisia, on niitä haastavampaa muuttaa jälkeenpäin.

Käyttöliittymän toiminnallisuus perustuu erityisesti JavaScript-koodiin, joka päivittää verkkosivuja dynaamisesti sekä lähettää pyyntöjä palvelimelle käyttäjän toimenpiteiden perusteella. Näkymät taas käsittelevät näitä pyyntöjä, kommunikoivat tietokannan kanssa mallien kautta, ja palauttavat vastaukset käyttäjälle

halutussa muodossa hahmottaen sisällön määritellylle HTML-sivulle. Kehitystyöni keskittyi Djangon arkkitehtuurin MVT-malliin kokonaisuutena ja näiden osien välillä tapahtuvaan vuorovaikutukseen ja siihen, miten vastuut jakautuvat. Testaus onnistui tehokkaasti Djangon sisäänrakennetun kehitysverkkopalvelimen avulla, kun koodin ja siihen tehdyt muutokset joko palvelimen tai käyttöliittymän puolella pystyi testaamaan nopeasti käytännössä, verkkoselainta päivittämällä.

Sovelluksen pääsivu

Kuvassa 4 on sovelluksen pääsivu ”index.html”, johon on sijoitettu kaikki sovelluksen päätoiminnot. Listaukset jo tallennetuista hevosista näkyvät sivulla, ja kuten kuvasta huomaa, tietokantaan on tallennettu kaksi hevosta. Hevosen nimeä klikkaamalla saa avattua hevosen tiedot. Painikkeesta ”Luo päiväkirjamerkintä” voi luoda uuden merkinnän, joka tallentuu hevosen päiväkirjaan. Hevosten profiileissa on linkit päiväkirjoihin, jotka aukeavat omille sivuilleen, jotta niiden tarkastelu ja merkintöjen muokkaus olisi sujuvaa. Pääsivu sisältää myös linkin erilliseen kalenteriin, joka myös avautuu omalle sivulleen.



Kuva 4. Sovelluksen pääsivu index.html.

Esimerkkikoodin 6 näkymä, Python-funktio ”index”, kysyy Horse-tietomallia hakeakseen kaikki mallin esiintymät tietokannasta eli kaikki tallennetut hevoset, ja palauttaa esiintymät HTTP-vastauksena pohjaan ”index.html”. Tämän perusteella kaikkien tallennettujen hevosten profiilit näkyvät pääsivulla. Myös näkymät tarvitsevat tuonteja tiedoston ”views.py” alkuun, kuten mallit tiedostossa ”models.py”, jotta Python tunnistaisi luokat, joita näkymät tarvitsevat toimiakseen. Esimerkiksi mallit tulee tuoda, jotta näkymät voivat pyytää tallennettuja tietoja tietokannasta tietyistä malleista. Olen lisännyt esimerkkikoodeihin tuonteja, jotka liittyvät tiettyyn näkymään, toistamatta kuitenkaan jo aiemmin tehtyjä tuonteja.

```
from .models import Horse, Diary, DiaryEntry, DiaryEntryFile, Events
from django.shortcuts import render


def index(request):
    horses = Horse.objects.all()
    return render(request, 'equiaiapp/index.html', {'horses': horses})
```

Esimerkkikoodi 6. Näkymä, joka kysyy Horse-tietomallia hakeakseen kaikki mallin esiintymät tietokannasta ja palauttaa vastauksen pohjaan index.html.

Pohja ”index.html”, jonka index-näkymä hahmontaa, on osittain kuvattu esimerkkikoodissa 7. HTML-elementti ’<ul>’ sisältää -elementin, jonne listautuu tietokantaan tallennetut hevoset. Django-tagi {% for horse in horses %} generoi HTML-sisältöä jokaiselle horses-muuttujan arvolle. Jokaiselle hevoselle muodostuu näin ollen oma tietue sivuille, aina kun uusi hevonen lisätään.

Esimerkkikoodista nähdään myös, että käyttäjän klikatessa hevosen nimeä, kutsutaan JavaScript-funktiota ”showHorseProfile”. Funktio lukee data-horse-id-attribuutin arvon linkin elementistä, sillä jokaiseen hevoseen liittyy yksilöllinen tunnus (horse_id). Funktio käyttää poimimaansa arvoa lähettäessään Ajax-pyynnön palvelimelle URL-osoitteeseen ”/horse_profile/{horseId}/”, joka ohjaa pyynnön näkymään ”horse_profile”, joka on tunnistettu hevosen tunnuksella.

```
<ul id="horse-list">
    {% for horse in horses %}
    <li>
        <div>
           <a href="javascript:void(0)"
            onclick="showHorseProfile(this)"
            data-horse-id="{{ horse.id }}">{{ horse.name }}</a>
        </div>
        <div id="horse-details" style="display: none;">
             <!—Hevosten tiedot listautuvat tänne. -->
        </div>

        <div class="button-link-group">
            <button class="create-diary-entry" data-horse-id="
             {{ horse.id }}">Luo päiväkirjamerkintä</button>

            <a href="/view_diary/{{ horse.id }}/" target="_blank"
            class="diary-link-style">Päiväkirjaan</a>
        </div>
    </li>
    {% endfor %}
</ul>
```

Esimerkkikoodi 7. HTML-elementti, joka listaa hevostietueet.

Edellisessä koodissa nähty JavaScript-funktion pyyntö ohjautuu esimerkkikoodin 8 näkymään ”horse_profile” URL-reitityksen mukaan. Näkymä kysyy Horse-mallilta tallennetut hevosesiintymät ja palauttaa tiedot JSON-muodossa, jolloin JavaScript käsittelee vastauksen ja luo HTML-sisältöä hevosen tiedoista. Tämä tapahtuu ilman sivun uudelleenlatausta, joka parantaa käyttökokemusta.

```
from django.http import JsonResponse


def horse_profile(request, horse_id):
    try:
        horse = Horse.objects.get(pk=horse_id)
        horse_data = {
            'id': horse.id,
            'name': horse.name,
            'gender': horse.gender,
            'birth_date': horse.birth_date.strftime('%Y-%m-%d')
             if horse.birth_date else None,
            'description': horse.description,
        }
        return JsonResponse({'success': True, 'horse': horse_data})
    except Horse.DoesNotExist:
        return JsonResponse({'success': False})
```

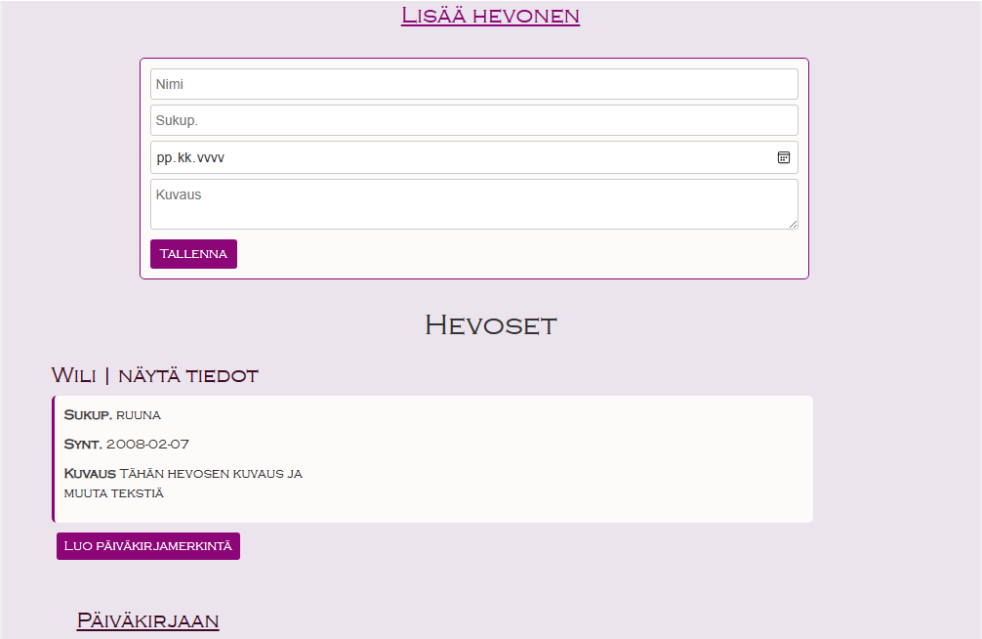
Esimerkkikoodi 8. Näkymä, joka vastaanottaa Ajax-pyynnön ja palauttaa vastauksen JSON-muodossa.

Käyttäjän klikatessa linkkiä ”Lisää hevonen”, toiminto kutsuu esimerkkikoodin 9 JavaScript-funktiota ”showAddHorseForm”, joka hakee HTML-elementin, jonka tunnus on ”add-horse-form”. Tällöin verkkosivulle avautuu lomake, ja koska CSS-näyttöominaisuuden arvoksi on määritelty ”block”, näkyy lomake lohkotason elementtinä ilman sivun uudelleen latautumista. Funktiolla ei ole tässä suoraa viestintää minkään näkymän kanssa, vaan se valmistelee käyttöliittymän käyttäjän syötettä varten. Tämä on hyvä esimerkki JavaScriptin ominaisuudesta, joka mahdollistaa sivun sisällön muokkaamisen ilman palvelimelta saatavaa tietoa, ”getElementById” on niin kutsuttu DOM-metodi.

```
function showAddHorseForm() {
    document.getElementById('add-horse-form').style.display = 'block';
}
```

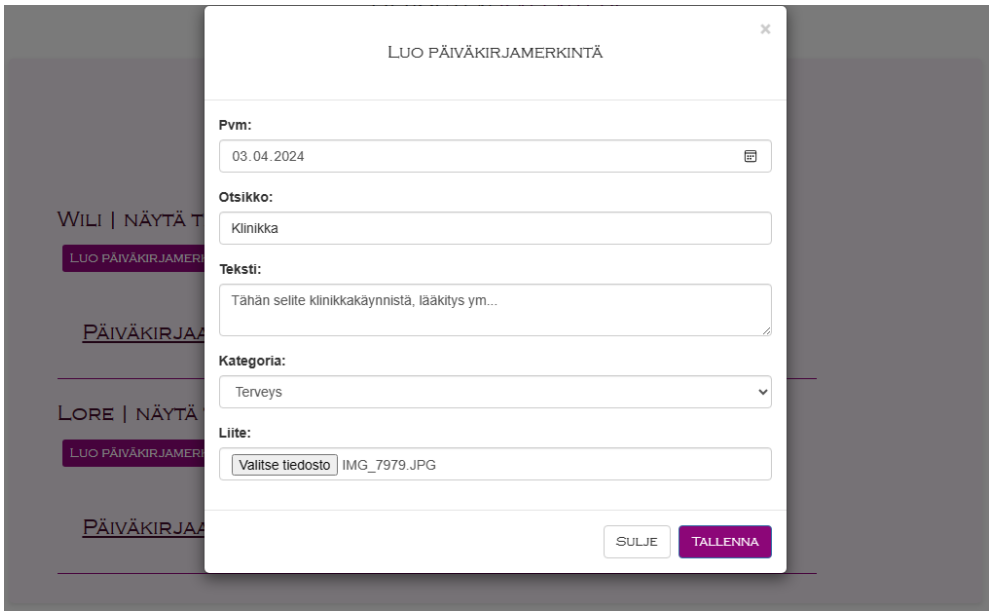
Esimerkkikoodi 9. Funktio, joka hakee HTML-elementin sen tunnuksella.

Kuva 5 näyttää sivuille avautuvan lomakkeen, jolla hevosen tiedot tallennetaan. Kun käyttäjä tallentaa lomakkeen, lomaketiedot lähetetään palvelimelle HTTP POST -pyynnöllä, ja palvelinpuolen koodi, näkymä ”add_horse”, käsittelee lomaketiedot. Näin ollen hevosen tiedot tallentuvat tietokantaan, ja uusi Horse-mallin esiintymä muodostuu. Kuvassa näkyy myös, miten hevosen tiedot sivulla näkyvät sen jälkeen, kun hevosen nimeä klikkaa.



Kuva 5. Lomake, jolla tallennetaan uusi hevonen tietokantaan. Hevosen tiedot tulevat näkyville nimeä klikatessa.

Hevosen profiilissa on painike ”Luo päiväkirjamerkintä”, jolla voi luoda hevoselle uuden päiväkirjamerkinnän. Kun käyttäjä klikkaa painiketta, verkkosivun päälle aukeaa modaalinen ikkuna, kuvassa 6, johon lomake tulee näkyviin. Käyttöliittymän kannalta lomakkeen avautuminen modaalina tai suoraan verkkosivuille, voi vaikuttaa käyttökokemukseen. Kun käyttää modaalista ikkunaa, käyttäjän huomio kohdistuu vain lomakkeeseen, koska se avautuu keskitetysti näytön päälle. Modaali kuitenkin voi olla haastavampi käyttää mobiililaitteilla, koska se voi peittää suuren osan näytöstä tai vaikeuttaa näytön vierittämistä.



Kuva 6. Käyttäjä voi luoda päiväkirjamerkinnän käyttäen modaali-ikkunaan avautuvaa lomaketta.

Kun käyttäjä tallentaa päiväkirjamerkinnän, kutsutaan JavaScript-funktiota ”submitDiaryEntry”, joka lähettää palvelimelle pyynnön merkinnän lisäämiseksi ja käyttöliittymän päivittämiseksi onnistuneen tallennuksen jälkeen. Esimerkkikoodi 10 näyttää pyynnön näkymän ”add_diary_entry” URL-osoitteelle käyttämällä fetch-funktiota, ja funktio jäsentää vastauksen rungon JSON-muodossa. Koodi havainnollistaa myös suojauksen, jossa JavaScript-funktio lähettää pyynnön näkymään, joka on suojattu @csrf_protect-dekoroinnilla. Tässä tapauksessa on varmistettava, että CSRF-token sisältyy pyyntöön. Tämä johtuu siitä, että @csrf_protect tarkistaa ainoastaan pyynnöt, jotka tulevat suoraan sivulta, eikä pyyntöjä, jotka lähetetään JavaScriptin kautta.

Kun JavaScriptiä käytetään lähettämään pyyntöjä palvelimelle ja käsittelemään vastauksia, olisi suositeltavaa käsitellä myös virheilmoitukset koodissa siten, että käyttäjä huomaa mahdollisen virhetilanteen. JavaScript voi käsitellä vain pyyntöjä ja vastauksia, jotka se saa palvelimelta, ja jos palvelin palauttaa virheen, se ei näy käyttäjälle selaimen oletusvirheilmoituksena. Tällöin JavaScript-koodiin tulee sisällyttää virheenkäsittely ja antaa käyttäjälle sopiva ilmoitus. Esimerkiksi, jos palvelin palauttaa JSON-virheraportin, JavaScriptin on tarkoitus

tulkita tämä vastaus. Esimerkkikoodin 10 funktio on tarkoitettu päiväkirjamerkinnän tallentamiseen, ja se näyttää käyttäjälle mahdollisen virheilmoituksen ponnahdusikkunassa, jos päiväkirjamerkinnän tallennus epäonnistuu.

```
function submitDiaryEntry() {
    var form = document.getElementById('diaryEntryForm');
    var formData = new FormData(form);
    var horseId = form.getAttribute('data-horse-id');

    var url = `/add_diary_entry/${horseId}/`;

    fetch(url, {
        method: 'POST',
        body: formData,
        headers: {
            'X-Requested-With': 'XMLHttpRequest',
            'X-CSRFToken': document.querySelector
            ('[name=csrfmiddlewaretoken]').value,
        }
    })
    .then(response => {
        if (!response.ok) {
            throw new Error('Pyyntö epäonnistui. Yritä uudelleen.');
        }
        return response.json();
    })
    .then(data => {
        if (data.success) {
            $('#diaryEntryModal').modal('hide');
            location.reload();
        }
    })
    .catch(error => {
        alert('Virhe: ' + error.message);
    });
}
```

Esimerkkikoodi 10. Fetch-funktio, joka lähettää pyynnön palvelimelle, ja palauttaa käyttäjälle virheilmoituksen, jos päiväkirjamerkinnän tallennus epäonnistuu.

Näkymä ”add_diary_entry”, johon edellisen JavaScript-funktion pyyntö ohjautuu, näkyy esimerkkikoodissa 11. Näkymä hakee ensin hevosen sen tunnisteen mukaan Horse-mallista ja luo uuden DiaryEntry-objektin, eli päiväkirjamerkinnän, ja tallentaa sen hevosen päiväkirjaan. Koodin alkuun on sijoitettu tuonteja, joita tämä näkymä tarvitsee, aiempien tuontien lisäksi. Suojausmerkintä @csrf_protect varmistaa, että POST-pyynnöt tarkistetaan CSRF-tokenin avulla, mikä estää mahdollisia CSRF-hyökkäyksiä. Vastaavasti pohjan lomake-elementin päiväkirjamerkinnän lisäämiseksi tulee sisältää {% csrf_token %} -tagi, mikä myös

on Djangon suositus CSRF-suojausta varten. Rivi ”@require_POST” taas varmistaa, että näkymä käsittelee vain POST-pyyntöjä.

```
from django.shortcuts import get_object_or_404
from django.views.decorators.csrf import csrf_protect
from django.views.decorators.http import require_POST


@csrf_protect
@require_POST
def add_diary_entry(request, horse_id):
    horse = get_object_or_404(Horse, id=horse_id)
    diary, created = Diary.objects.get_or_create(horse=horse)

    entry = DiaryEntry(diary=diary)
    entry.date = request.POST.get('date')
    entry.title = request.POST.get('title')
    entry.content = request.POST.get('content')
    entry.category = request.POST.get('category')
    if 'file' in request.FILES:
        entry.file = request.FILES['file']
    entry.save()

    return JsonResponse({'success': True, 'entry_id': entry.id})
```

Esimerkkikoodi 11. Näkymä hakee hevosen Horse-mallista tunnisteen perusteella ja luo uuden DiaryEntry-objektin, eli päiväkirjamerkinnän.

Päiväkirja

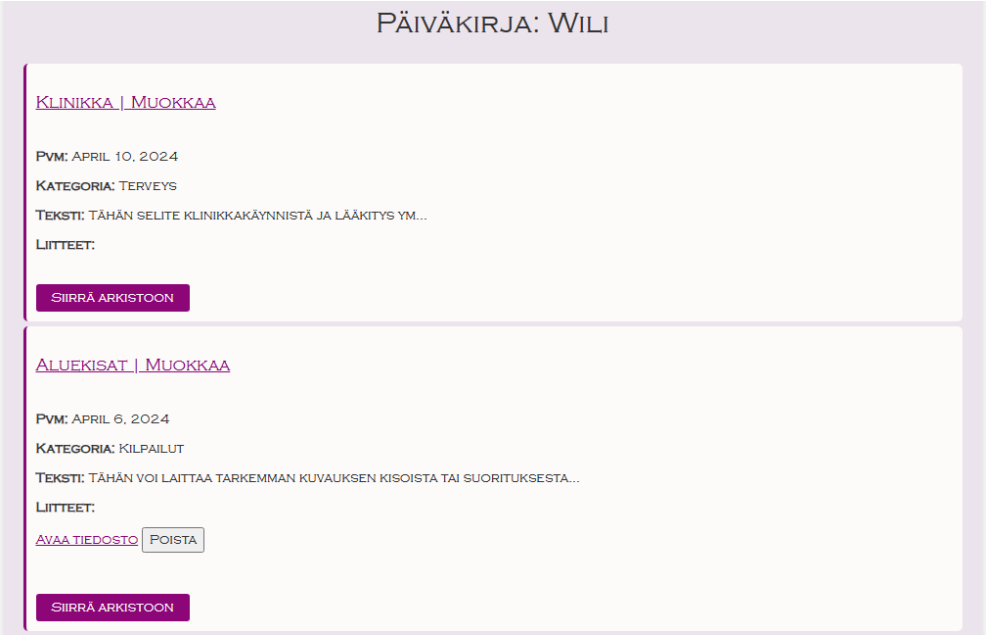
Hevosten päiväkirjat avautuvat omille sivulleen hevosen tunnisteen mukaan, kun hevosen profiilissa klikkaa linkkiä “Päiväkirjaan”. Näkymä ”view_diary”, vastaa tietyn hevosen DiaryEntry-esiintymien, eli päiväkirjamerkintöjen, näyttämisestä. Näkymä palauttaa kaikki päiväkirjamerkinnät, joita ei ole vielä arkistoitu. Kun tämä näkymä vastaa pohjan ”view_diary.html” hahmontamisesta sivulle, näkymä ”get_diary_entries” taas palauttaa JSON-vastauksen, joka sisältää tiedot jokaisesta hevosen päiväkirjamerkinnästä jäsennellyssä muodossa. Esimerkkikoodin 12 palauttama vastaus sisältää sanakirjatietorakenteen ”entries_data” kaikkien päiväkirjamerkintöjen tiedoista.

```
def get_diary_entries(request, horse_id):
    horse = get_object_or_404(Horse, id=horse_id)
    entries = DiaryEntry.objects.filter(diary__horse=horse)
    .order_by('-date')
    entries_data = [{
        'id': entry.id,
        'date': entry.date.isoformat(),
        'title': entry.title,
        'content': entry.content,
        'category': entry.category,
        'file_url': entry.file.url if entry.file else None
    } for entry in entries]

    return JsonResponse({'success': True, 'entries': entries_data})
```

Esimerkkikoodi 12. Näkymä palauttaa JSON-vastauksen, joka sisältää tiedot jokaisesta hevosen päiväkirjamerkinnästä.

Kuvassa 7 näkyy hevosen päiväkirja sivulla ”view_diary.html”, hevosen tunnuksen mukaan. Päiväkirjaan on tallennettu kaksi merkintää, joista toiseen on tallennettu liite. Liite on merkinnän yhteydestä mahdollista poistaa ilman että merkintää tarvitsee muuten päivittää. Näkymä ”delete_diary_entry_file” vastaa tiettyyn päiväkirjamerkintään liittyvän tiedoston poistamisesta, joka on tunnistettu sen tunnuksella (file_id), ja päivittää siihen liittyvän DiaryEntryFile-esiintymän.



Kuva 7. Hevosen päiväkirja, jossa kaksi tallennettua päiväkirjamerkintää.

Käyttäjä voi muokata tai poistaa päiväkirjamerkintöjä sekä lisätä liitteitä klikkaamalla merkinnän otsikkoa. Kun kaksi aiempaa näkymää käsittelevät tietyn hevosen kaikkien päiväkirjamerkintöjen hakemista ja näyttämistä sivuilla, näkymä ”get_diary_entry” noutaa tietyn DiaryEntry-esiintymän, eli päiväkirjamerkinnän, tiedot sen yksilöllisen tunnisteen (entry_id) perusteella. Täten käyttäjä saa avattua tietyn päiväkirjamerkinnän ja pystyy muokkaamaan sitä.

Käyttäjän klikatessa päiväkirjamerkinnän otsikkoa, toiminto kutsuu esimerkkikoodin 13 funktiota ”showDiaryEntryModal”, joka avaa modaali-ikkunan merkinnän mahdollista muokkaamista varten. Funktio ottaa tunnisteen (entryId) parametriksi ja lataa olemassa olevat syöttötiedot lomakkeeseen ”diaryEntryForm”. Merkinnän tiedot haetaan URL-osoitteesta ”/get_diary_entry/${entryId}/”, joka viittaa edellä esiteltyyn näkymään. Onnistuneen vastauksen jälkeen se asettaa lomakkeen kenttiin palvelimelta vastaanotetut tiedot.

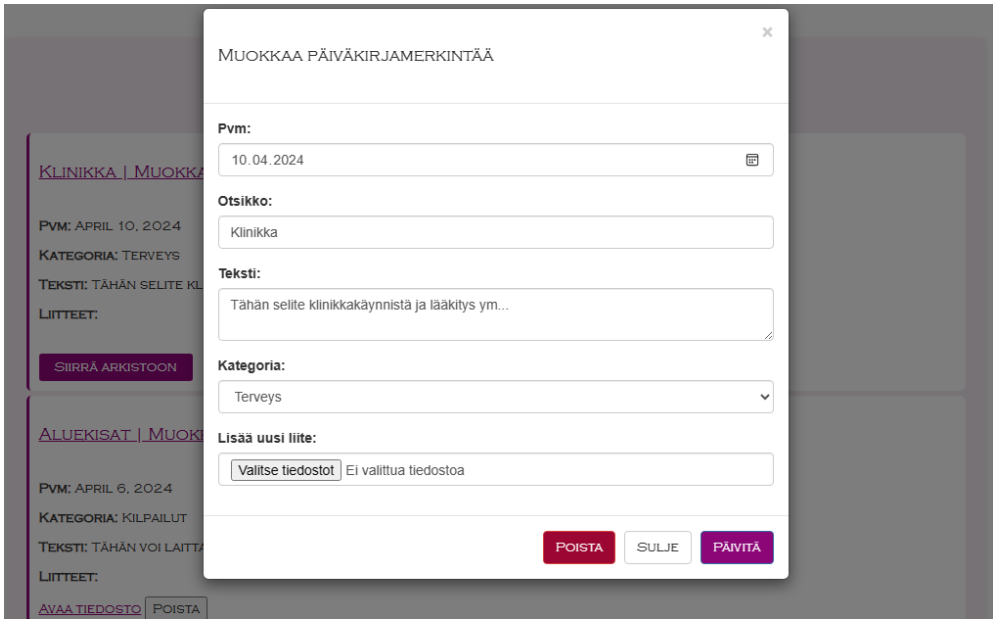
```
function showDiaryEntryModal(entryId = null) {
    var form = document.getElementById('diaryEntryForm');
    form.reset();
    document.getElementById('entryId').value = entryId || '';

    if (entryId) {
        fetch(`/get_diary_entry/${entryId}/`)
            .then(response => {
                if (!response.ok) {
                    throw new Error('Virhe. Yritä uudelleen.');
                }
                return response.json();
            })
            .then(data => {
                form.elements['date'].value = data.entry.date;
                form.elements['title'].value = data.entry.title;
                form.elements['content'].value = data.entry.content;
                form.elements['category'].value = data.entry.category;

                $('#diaryEntryModal').modal('show');
            })
            .catch(error => {
                alert('Virhe: ' + error.message);
            });
    } else {
        $('#diaryEntryModal').modal('show');
    }
}
```

Esimerkkikoodi 13. Funktio, joka avaa modaali-ikkunan päiväkirjamerkinnän tarkastelua ja mahdollista päivittämistä varten.

Kuva 8 näyttää tallennetun päiväkirjamerkinnän lomakkeen, jossa voi päivittää tietoja sekä lisätä uuden liitetiedoston. Alkuperäisen merkinnän yhteyteen mahdollisesti lisätty liite ei näy lomakkeella, vaan ainoastaan sivulla, josta se oli mahdollista avata tai poistaa. Liitteiden hallintaa voisi jatkossa parantaa, kuten mahdollistaa niiden lisääminen suoraan sivulla, johon ne tallentuvat. Käyttäjän napsauttaessa ”Päivitä”, funktio ”updateDiaryEntry” lähettää palvelimelle Ajax-pyynnön merkinnän päivittämiseksi. Näkymä ”update_diary_entry” päivittää kyseisen DiaryEntry-esiintymän annetuilla tiedoilla, ja mahdollistaa tietyn päiväkirjamerkinnän päivitettyjen tietojen lähettämisen tietokantaan.



Kuva 8. Lomake, jossa käyttäjä voi muokata päiväkirjamerkintää tai poistaa sen.

Jos käyttäjä napsauttaa päiväkirjamerkinnän Poista-painiketta, kutsutaan JavaScript-funktiota ”deleteDiaryEntry”, joka lähettää DELETE-pyynnön palvelimelle. Näkymä ”delete_diary_entry” käsittelee pyynnön poistaakseen merkinnän tietokannasta. Esimerkkikoodin 14 näkymä on suunniteltu käsittelemään HTTP DELETE -pyyntöjä, joilla poistetaan sen tunnisteella (entry_id) tunnistettu päiväkirjamerkintä DiaryEntry-esiintymistä. Tämä metodi varmistaa, että vain kyseiset pyynnöt saavat käyttää tätä näkymää.

```
from django.views.decorators.http import require_http_methods


@csrf_protect
@require_http_methods(["DELETE"])
def delete_diary_entry(request, entry_id):
    try:
        entry = DiaryEntry.objects.get(id=entry_id)
        entry.delete()
        return JsonResponse({'success': True})
    except DiaryEntry.DoesNotExist:
        return JsonResponse({'success': False, 'error':}, status=404)
```

Esimerkkikoodi 14. Näkymä välittää poistopyynnön, jolla päiväkirjamerkintä poistuu tietokannasta, eli DiaryEntry-esiintymistä.

URL-reititystä käyttöliittymän ja palvelimen välillä on helpompi hahmottaa esimerkkikoodin 15 avulla, joka näyttää pienen osan sovelluksen urls.py-tiedostoon määritellyistä URL-osoitteista. Kukin URL-osoite yhdistyy sitä vastaavaan näkymään, ja sisältää tarvittaessa tunnisteen (id).

```
from django.urls import path
from .views import add_diary_entry, view_diary, update_diary_entry
from django.conf import settings

urlpatterns = [
    path('add_diary_entry/<int:horse_id>/',
          add_diary_entry, name='add_diary_entry'),
    path('view_diary/<int:horse_id>/', view_diary, name='view_diary'),
    path('update_diary_entry/<int:entry_id>/',
          update_diary_entry, name='update_diary_entry'),
]
```

Esimerkkikoodi 15. Tiedoston ”urls.py” URL-osoitteita, joiden mukaan pyynnöt ohjautuvat oikeille näkymille.

Arkisto

Päiväkirjamerkinnät voi arkistoida sen jälkeen, kun tarvetta merkinnän päivittämiselle tai litteiden lisäämiselle ei enää ole. Päiväkirjamerkinnän yhteydessä on painike ”Siirrä arkistoon”. Arkisto on sijoitettu samalle sivulle kuin hevosen päiväkirja, sivun alalaitaan. Kun päiväkirjamerkinnän siirtää arkistoon, merkinnät tallentuvat eri kategorioiden alle, jolloin tietojen hallinta ja etsintä on jälkeenpäin helpompaa. Kuten päiväkirja, myös arkisto on hevoskohtainen.

Käyttäjän arkistoidessa päiväkirjamerkinnän, esimerkkikoodin 16 funktio ”archiveDiaryEntry” lähettää palvelimelle Ajax-pyynnön merkinnän arkistointia varten. Funktion parametri ”entryId” viittaa arkistoitavan päiväkirjamerkinnän tunnisteeseen. Näkymä ”archive_diary_entry” merkitsee tunnistetun päiväkirjamerkinnän arkistoiduksi, jolloin is_archived-kentän arvoksi asetetaan ”True”. Näkymä palauttaa JSON-vastauksen ja muutokset tallennetaan tietokantaan.

```
function archiveDiaryEntry(entryId, buttonElement) {
    var csrftoken = $('meta[name="csrf-token"]').attr('content');

    if (confirm('Haluatko varmasti siirtää arkistoon?')) {
        $.ajax({
            url: `/archive_diary_entry/${entryId}/`,
            type: 'POST',
            headers: { "X-CSRFToken": csrftoken },
            success: function(response) {
                if (response.success) {
                    buttonElement.closest
                    ('.diary-entry-container').remove();
                    alert('Merkintä arkistoitu onnistuneesti.');
                } else {
                    alert('Virhe siirrettäessä arkistoon.');
                }
            },
            error: function(xhr, status, error) {
                alert('Virhe tapahtuman käsittelyssä.');
            }
        });
    }
}
```

Esimerkkikoodi 16. JavaScript-funktio, joka lähettää palvelimelle Ajax-pyynnön päiväkirjamerkinnän arkistointia varten.

Esimerkkikoodin 17 näkymä ”filter_archived_entries” suodattaa arkistossa olevat päiväkirjamerkinnät arvojen ”horse_id” ja ”category” perusteella, eli sen kategorian mukaan, joka on päiväkirjamerkinnän tallennuksen yhteydessä valittu. Kun käyttäjä napsauttaa arkistossa tiettyä kategoriaa, kutsutaan JavaScript-funktiota ”fetchArchivedEntries”, joka lähettää Ajax GET -pyynnön näkymän URL-osoitteeseen ”/filter_archived_entries/${horseId}/${category}/”, noutaakseen arkistoidut merkinnät. Näkymä palauttaa vastauksen, joka sisältää luettelon arkistoiduista merkinnöistä. Se käyttää DjangoJSONEncoder-komentoa varmistaakseen, että kaikki tiedot on sarjoitettu oikein JSON-muotoon, koska jotkin tietotyypit eivät välttämättä ole suoraan sarjoitettavissa.

```
from django.core.serializers.json import DjangoJSONEncoder


def filter_archived_entries(request, horse_id, category):
    horse = get_object_or_404(Horse, id=horse_id)

    archived_entries = DiaryEntry.objects.filter
    (diary__horse=horse, is_archived=True, category=category)

    entries_list = [{
        'id': entry['id'],
        'title': entry['title'],
        'date': entry['date'].strftime('%Y-%m-%d'),
        'content': entry['content'],
        'category': entry['category']
    } for entry in archived_entries.values('id', 'title',
    'date', 'content', 'category')]

    return JsonResponse({'entries': entries_list})
```

Esimerkkikoodi 17. Näkymä, joka suodattaa arkistoidut päiväkirjamerkinnät kategorian mukaan.

Kuva 9 näyttää Terveys-kategoriaan arkistoidut päiväkirjamerkinnät. Kun käyttäjä klikkaa merkinnän otsikkoa, JavaScript-funktio ”showArchivedEntryModal(entryId)” käyttää Ajax-kutsua hakeakseen arkistoidun päiväkirjamerkinnän tiedot palvelimelta. Näkymä ”get_archieved_entry” hakee tiedot tietystä arkistoidusta päiväkirjamerkinnästä ”entry_id” mukaan sekä siihen liittyvät tiedostot. Päiväkirjamerkinnän tiedot avautuvat modaali-ikkunaan katseluominaisuudessa.

Kuva 9. Päiväkirjamerkinnät arkistoituvat kategorioittain.

Kuten kuvista huomaa, sivujen sekä lomakkeiden tyyli on yhtenäinen. Olen linkittänyt HTML-pohjiin Bootstrap- sekä CSS-tyylikomponentteja sisältäviä kirjastoja. Linkitykset sisältävät myös jQueryn, joka yksinkertaistaa HTML-dokumenttien läpikulkua, tapahtumien käsittelyä ja Ajax-vuorovaikutusta. Sivujen ulkoasun, kuten asettelun, fontit, värit sekä painikkeiden muotoilun olen tyylitellyt CSS:n avulla pohjissa, jotta verkkosivut olisivat visuaalisuudeltaan samanlaiset. Mikäli haluaa sivuista uniikimman näköisen, CSS tulee kirjoittaa itse.

Kalenteri

Ulkoisten kirjastojen linkityksen avulla on toteutettu myös hevosten yhteinen kalenteri. Kalenteri on sisällytetty sovellukseen JavaScript-kirjastolla ”FullCalendar”. Tarkemmin se on JavaScript-kalenterilaajennus, jota käytetään tapahtumien näyttämiseen ja hallintaan verkkosovelluksissa. FullCalendar tarjoaa mukautettavia ja interaktiivisia kalenterinäkymiä, kuten päivä-, viikko-, kuukausinäkymät. Tapahtumia voidaan kirjata klikaten kalenteripäivää sekä muokata raahaamalla tapahtuma eri päivälle tai muuttamalla sen kokoa.

Fullcalendar, kuten sen edellyttämät kirjastot Moment.js, JQuery, Bootstrap, ovat tuotu linkkeinä pohjaan ”appointments.html”. Kalenterinäkymä avautuu tälle verkkosivulle, kun käyttäjä pääsivulla klikkaa linkkiä ”Kalenteri”. Moment.js on JavaScript-kirjasto päivämäärien ja aikojen jäsentämiseen, käsittelyyn ja muotoiluun. FullCalendar tarvitsee myös CSS-tiedoston, joka sisältää tyylisivut FullCalendar-laajennuksen hahmontamiseen oikein.

Hevosten kalenterin toiminnallisuus perustuu Events-malliin, ja esimerkkikoodin 18 näkymä ”appointments” hahmontaa kaikki tietokantaan tallennetut Events-esiintymät, eli tallennetut tapahtumat, näkyville kalenteriin. Koska Events-malli oli linkitetty Horse-malliin, hakee näkymä myös kaikki hevosesiintymät. Linkityksen tein sen vuoksi, että kalenteritapahtuman voisi tarvittaessa kohdistaa tietylle hevoselle. Kalenteri ei kuitenkaan ole yhteydessä Diary-malliin, eli hevosten päiväkirjoihin.

```
def appointments(request):
    all_events = Events.objects.all()
    horses = Horse.objects.all()
    context = {
        "events": all_events,
        "horses": horses,
    }
    return render(request, 'equiaiapp/appointments.html', context)
```

Esimerkkikoodi 18. Näkymä, joka hahmontaa verkkosivulle kaikki tietokantaan tallennetut tapahtuma- ja hevosesiintymät.

Esimerkkikoodin 19 näkymä ”all_events” on suunniteltu palauttamaan tiedot yksittäisistä kalenteritapahtumista JSON-vastauksena. For-silmukan sisällä tapahtuman nimi tallennetaan title-muuttujaan. If-ehto tarkistaa, liittyykö tapahtuma tiettyyn hevoseen. Tällöin hevosen nimi liitetään tapahtuman otsikkoon.

```
def all_events(request):
    all_events = Events.objects.select_related('horse').all()
    out = []
    for event in all_events:
        title = event.name
        if event.horse:
            title += f" | {event.horse.name}"
        out.append({
            'title': title,
            'description': event.description,
            'id': event.id,
            'start': event.start.strftime("%Y-%m-%d %H:%M:%S"),
            'end': event.end.strftime("%Y-%m-%d %H:%M:%S"),
        })
    return JsonResponse(out, safe=False)
```

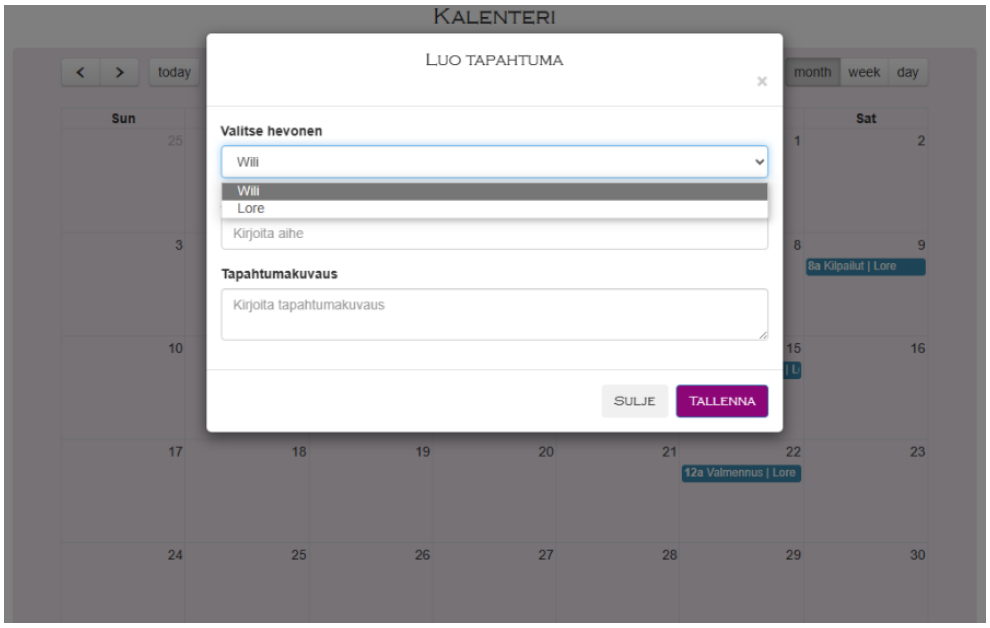
Esimerkkikoodi 19. Näkymä, joka palauttaa yksittäisten kalenteritapahtumien tiedot.

Kun käyttäjä klikkaa kalenterista päivää, johon hän haluaa tapahtuman luoda, se käynnistää JavaScript-funktion, jolloin modaali-ikkunaan avautuu lomake, jossa käyttäjä voi lisätä uuden tapahtuman. Kun käyttäjä tallentaa tapahtuman, funktio lähettää tiedot palvelimelle esimerkkikoodin 20 Ajax-pyynnöllä, päivittää kalenterin ja ilmoittaa käyttäjälle tapahtuman lisäämisestä tai mahdollisesta virheestä. Näkymä ”add_event” käsittelee tapahtumatietojen vastaanottamisen POST-pyynnöstä.

```
$.ajax({
    type: "POST",
    url: '/add_event/',
    data: {
        'csrfmiddlewaretoken': '{{ csrf_token }}',
        'title': title,
        'description': description,
        'start': startFormatted,
        'end': endFormatted,
        'horse_id': horseId
    },
    dataType: "json",
    success: function(response) {
        calendar.fullCalendar('refetchEvents');
        alert("Tapahtuma lisätty onnistuneesti.");
        $('#eventModal').modal('hide');
    },
    error: function(xhr, status, error) {
        alert("Virhe: " + xhr.responseText);
    }
});
```

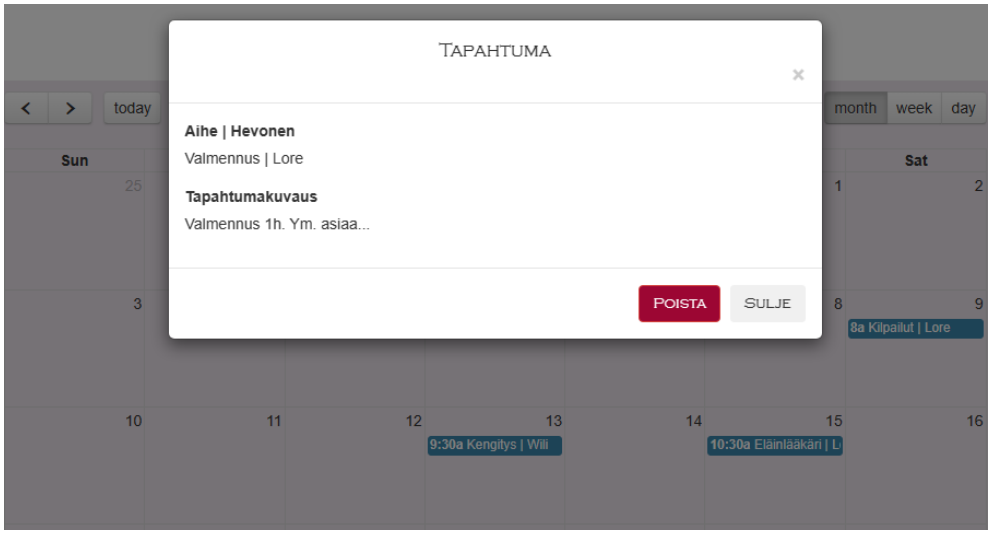
Esimerkkikoodi 20. AJAX-pyyntö kalenteritapahtuman tallentamiseksi.

Kuva 10 näyttää lomakkeen, jossa tapahtuma tallennetaan. Näkymä ”get_horses” hakee kaikki Horse-esiintymät, jolloin hevoset ovat valittavissa.



Kuva 10. Kalenteritapahtuman kohdistus tietylle hevoselle.

Kun käyttäjä napsauttaa tiettyä tallennettua tapahtumaa kalenterissa, se avaa modaalin, kuvassa 11. Modaalista näkee tapahtumakuvauksen, ja tarvittaessa tapahtuman voi poistaa. Näkymä ”remove_event” poistaa tapahtuman sen event_id:n perusteella. Kalenteritapahtumien muokkaamiseksi on kaksi JavaScript-funktiota. Molemmat funktiot lähettävät Ajax-kutsut palvelimelle, näkymään ”/update_event/”, niiden yksilöllisten tunnusten (eventId) mukaan. Funktioiden avulla kalenteritapahtuman voi raahata eri päivälle, tai muuttaa tapahtuman kokoa, eli rajaamalla tapahtuman alkamis- ja päättymisaikoja.



Kuva 11. Tapahtumakuvauksen näkee klikkaamalla tapahtumaa kalenterissa.

Kuten kalenterin ulkoasusta nähdään, kalenteripohjan tyyli tulee suoraan FullCalender:n kirjastoista. CSS-tyylit olen kuitenkin lisännyt appointments.html-pohjaan samaan tapaan kuin muihin sovelluksen pohjiin, sekä linkittänyt samat Bootstrap- sekä CSS-tiedostot, jotta koko sivusto olisi värimaailmaltaan, asettelultaan sekä modaalien tyyliltä samankaltainen. Kalenteritoimintojen logiikka on määritelty näkymien avulla ja käyttöliittymän toiminnallisuus JavaScriptin avulla, kuten muissakin sovelluksen toiminnoissa. Kalenteri oli hyvä täydennys sovellukseen, sillä sen avulla on mahdollista hallita ja tarkastella helposti tulevia tapahtumia, ja varmistaa ettei aikataulunsuunnittelussa tule päällekkäisyyksiä.

4 Tulokset

Työn tarkoitus oli suunnitella ja toteuttaa verkkosovellus hevosten ylläpitoon liittyvien tietojen hallintaan keskitetysti. Kuten työn tavoitteena oli, sovellus sisältää tarvittavat toiminnallisuudet hevosten päiväkirjan ylläpitämiseen kategorioittain, liitetiedostojen tallentamiseen, merkintöjen arkistointiin sekä tapahtumien aikataulutukseen. Sovelluksen taustajärjestelmän, sekä palvelinpuolen ja käyttöliittymän toimintojen vuorovaikutus oli vaivatonta suunnitella ja toteuttaa Djangon projektirakenteen sekä arkkitehtuurimallin avulla. Sain toteutettua toimivan ja helppokäyttöisen sovelluksen, rajaamalla sovellukseen vain käyttötarkoitukseen liittyvät olennaisimmat toiminnot, joita eri käyttöliittymätekniikat tukivat.

Mallien avulla määrittelin tietokannan rakenteen ja näkymät toimivat siltana tietokannan ja käyttöliittymätekniikoiden välillä. JavaScript-koodin asynkroniset tekniikat auttoivat tekemään käyttöliittymän toiminnoista interaktiivisempia, jolloin sovelluksen käyttö osoittautui tehokkaaksi ja sujuvaksi käyttäjän näkökulmasta. JavaScript mahdollisti reaaliaikaisen tiedonsiirron palvelimen ja selaimen välillä, ilman sivun uudelleenlatausta, mikä nopeutti sivuston reagointia ja vastauksia käyttäjän pyyntöihin. Sovelluksen eri osien toiminnallisuudet sekä palvelimen että käyttöliittymän puolella, eli koodi kokonaisuudessaan sekä muutokset koodissa, oli sujuvaa testata reaaliaikaisesti Djangon kehityspalvelimen avulla. Virhetilanteet ja puutteet tietyissä toiminnoissa oli helppoa korjata vaikuttamatta koodiin muissa sovelluksen osa-alueissa.

Ulkoiset kirjastot, kuten CSS-kehys Bootstrap, tarjosivat valmiita komponentteja muun muassa lomakkeiden tyylittelyyn. Tämä auttoi sivujen yhtenäistämistä ja ulkoasun suunnittelua. Verkkosivujen tyylin ja asettelun sekä värit ja fontit täydensin CSS-koodin avulla. Täten myös käyttöliittymän visuaaliset ominaisuudet tarjoavat miellyttävän käyttökokemuksen. Hyödynsin sovellukseeni myös JavaScript-kirjastoa FullCalender joka tarjoaa valmiin kalenteripohjan. Tämä täydensi mukavasti sovelluksen kokonaisuutta ja tarjosi tarpeellisen työkalun aikataulunhallintaan sisältäen valmiita rajapintoja eri toiminnallisuuksiin.

5 Johtopäätökset

Tulokset osoittavat, että Djangon projektirakenne ja tekniikat mahdollistavat sovelluksen sujuvan kehityksen ja ylläpidon. Kokonaisuus pysyi koko sovelluskehityksen ajan hyvin hallinnassa sekä palvelimen että käyttöliittymän puolella, ja yksityiskohtaiset ominaisuudet oli mahdollista toteuttaa joustavasti. Sovelluksen kehitysprosessi oli sujuva, ja virheiden korjaaminen oli vaivatonta reaaliaikaisen toimintojen testauksen avulla. Työn toteutukseen valittu Django-verkkokehys toteutti odotukseni siitä miten loogista ja suoraviivaista sen avulla oli toteuttaa verkkosovellus.

Python-ohjelmointikielen selkeys ja helppo luettavuus auttoivat jäsentämään koodin siten, että se olisi mahdollisimman ymmärrettävää ja vaivatonta ylläpitää. Djangon arkkitehtuurimalli ja siihen yhdistetyt käyttöliittymätekniikat mahdollistivat toimivan sovellusrakenteen ja toimintalogiikan toteuttamisen, ja vain tarpeelliset ominaisuudet ja tärkeät toiminnallisuudet oli helppoa rajata. Sovelluksen käyttökokemus vastasi sitä millaisen käytettävyyden ja suorituskyvyn sovelluksella tavoittelin olevan, siten että myös luotettavuus ja turvallisuus toteutuvat.

Työlle asetetut tavoitteet saavutettiin ja lopputuloksena on toimiva verkkosovellus hevosten tietojen ylläpitoon sekä ajankäytön hallintaan. Tämä mahdollistaa hyvän pohjan sovelluksen jatkokehitykselle ja laajentamiselle. Sovellukseen voisi lisätä toimintoja monipuolisemmin ja mahdollisuuden tallentaa yksityiskohtaisemmin eri tietoja. Suorituskykyä ja käyttökokemusta voisi edelleen kehittää. Käyttöliittymän estetiikkaa voisi parantaa sekä varmistaa sovelluksen skaalautuvuus eri laitteilla ja selaimilla. Työn tuloksia voidaan hyödyntää monipuolisesti, koska toiminnallisuudet ovat helposti muunnettavissa muunkin tyyppiseen tiedon tallentamiseen ja hallintaan vastaavanlaisissa projekteissa.

Lähteet

Buelta, Jaime. 2022. Python Architecture Patterns. E-kirja. Packt Publishing.

Chidera, Edeh Israel. 2023. How Functions Work in JavaScript. Verkkoaineisto. freeCodeCamp. <https://www.freecodecamp.org/news/understanding-functions-in-javascript/>. Luettu 1.4.2024.

Cross Site Request Forgery protection. Verkkoaineisto. Django Software Foundation. <https://docs.djangoproject.com/en/3.2/ref/csrf/>. Luettu 16.3.2024.

Dinder, Michael. 2022. Becoming an Enterprise Django Developer. E-kirja. Birmingham Packt Publishing.

Django introduction. Verkkoaineisto. MDN Web Docs. <https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Introduction>. Päivitetty 28.2.2024. Luettu 2.3.2024.

Django Models. Verkkoaineisto. GeeksforGeeks. <https://www.geeksforgeeks.org/django-models/>. Päivitetty 5.4.2024. Luettu 12.4.2024.

Django Web Framework (Python). Verkkoaineisto. MDN Web Docs. <https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django>. Päivitetty 3.7.2023. Luettu 3.3.2024.

Frontend vs Backend. Verkkoaineisto. GeeksforGeeks. <https://www.geeksforgeeks.org/frontend-vs-backend/>. Päivitetty 18.4.2023. Luettu 2.3.2024.

General Python FAQ. Verkkoaineisto. Python Software Foundation. <https://docs.python.org/3/faq/general.html>. Päivitetty 16.4.2024. Luettu 17.4.2024.

Ghidersa, Mihaela Roxana. 2023. Software Architecture for Web Developers. E-kirja. Packt Publishing.

Guta, Gabor. 2022. Pragmatic Python Programming. E-kirja. Berkeley, CA Apress L. P.

Introduction to the server side. Verkkoaineisto. MDN Web Docs. <https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Introduction>. Päivitetty 1.1.2024. Luettu 2.3.2024.

JavaScript Fetch Vs Ajax. 2023. Verkkoaineisto. Medium. <https://medium.com/@bin61615/javascript-fetch-vs-ajax-c3506187fd31>. Luettu 13.4.2024.

JavaScript HTML DOM. Verkkoaineisto. W3Schools. <https://www.w3schools.com/js/js_htmldom.asp>. Luettu 13.2.2024.

JSON – Introduction. Verkkoaineisto. W3Schools. <https://www.w3schools.com/js/js_json_intro.asp>. Luettu 14.3.2024.

Myers, Dominic. 2020. Front-End Developer. E-kirja. BCS, The Chartered Institute for IT.

Shaw, Ben; Badhwar, Saurabh; Guest, Chris & S, Bharath Chandra K. 2023. Web Development with Django. E-kirja. Packt Publishing.

Setting up Visual Studio Code. Verkkoaineisto. Visual Studio Code. <https://code.visualstudio.com/docs>. Päivitetty 7.3.2024. Luettu 12.3.2024.

Simpson, Jonathon. 2023. How JavaScript Works. E-kirja. Springer Nature.

Top 7 Programming Languages for Backend Web Development. Verkkoaineisto. GeeksforGeeks. <https://www.geeksforgeeks.org/top-7-programming-languages-for-backend-web-development/>. Päivitetty 20.3.2024. Luettu 17.4.2024.

TIOBE Index for April 2024. Verkkoaineisto. TIOBE Software BV. <https://www.tiobe.com/tiobe-index/>. Luettu 3.4.2024.

Writing your first Django app, part 1. Verkkoaineisto. Django Software Foundation. <https://docs.djangoproject.com/en/5.0/intro/tutorial01/>. Luettu 13.3.2024.

Writing your first Django app, part 2. Verkkoaineisto. Django Software Foundation. <https://docs.djangoproject.com/en/5.0/intro/tutorial02/>. Luettu 13.3.2024.